JoshJava to MARIE Complier

Joshua Zachariah (Grade 11) and Jeff Edmonds (CS Prof[1])

Abstract

We describe what we learned while writing a compiler from a restriction of Java to MARIE assembler code.

1 Introduction

Apology: Let us start by apologizing. Neither of us have taken a course or read anything on compilers or on MARIE. In our usual way, we prefer to piece it together ourselves. But we hope, this will give something that will be easy to follow by other novices. Perhaps someone will find some of it useful for a course.

A Parser: My favorite algorithm of all time is a look-ahead-one parser. A context free grammar uses rules like

exp $\Rightarrow$ term + term + ... + term
term $\Rightarrow$ fact * fact * ... * fact

to describe how an object like an expression $x+y\times z$ consists of the sum of terms x and $y\times z$, while the term $y\times x$ consists of the product of factors y and z. The parsing algorithm takes such an expression as input and builds a tree showing its structure. It gets a routine $GetExp$ to parse an expression $x+y\times z$ by recursively asking $GetTerm$ to parse its terms x and $y\times z$. Then $GetExp$ builds the parse tree of this expression from the parse trees of the terms given to it by $GetTerm$. I teach this in my third year and grad algorithms courses. When my son Josh needed to do a final project for grade 11 CS, I was excited to do something here.

A Complier: A compiler translates something from one language into another, for example, a Java program into machine code. Typically it makes two passes through the Java program. The first builds a symbol table explaining about all of the variable names, their types, and their scoping. The second pass parses the Java program building a parse tree describing the nested structures within the Java program. As this parse tree is built, it translates each little Java structure into the corresponding machine code structure. Having avoided the compiler course both as an undergrad and a graduate student because of the amount of coding required (and I turned down a job at IBM compiler group), I was inspired now to write a compiler. Both for ease of programming and for aesthetic reasons, I wanted it to be a one pass memoryless parser, meaning that it simply reads through the Java program once, parses it using a grammar, while spewing out the corresponding machine code.

JoshJava: We started by restricting Java to make our task easier and to allow it to be compiled using one pass. At first, we thought that each routine must have the same fixed variable set all of type integers. But later we were able to have arbitrary sets of variable names. The only restriction was now is that global variables need to start with a capital and local variables with a small letter.

MARIE: Our friend Gara told us about a very simple assembly language that is used in class rooms. It comes with a graphical simulator/debugger and a graphical Data Path simulator. We decided to compile JoshJava into this. Its extremely limited instruction set made the task even more of a fun puzzle.

Surprisingly Easy: Building this compiler required dealing with more details than I initially anticipated, but conceptually was far easier than I expected. The steps are as follows.

[1]York University, Canada. jeff@cs.yorku.ca. Supported in part by NSERC Canada.

Grammar: The first step is to design a context free grammar to describe all the structures within JoshJava code.

Parser: Writing a parser using a grammar is quite straight forward. There is a routine for each type of structure which simply follows the rules of the grammar.

Corresponding MARIE code: As each structure within the JoshJava code is found, the corresponding MARIE code is produced.

One Step at a Time: The key to not getting overwhelmed is to not think about writing an entire compiler, but to focus on one structure at time, trusting that the other structures get handled magically on their own.

Three Computations: The only complex thing is that there are three computation that you need to keep track of in your brain. You need to see the link between them but not to mix them up.

JoshJava: The computation of executing the JoshJava code.

Compiler: The computation of the compiler (also written in Java)

MARIE code: The computation of executing the resulting MARIE code.

What makes understanding these three computations easier is that each can be broken down into the exact same tree of structures.

1.1 Outline:

This Document: This document contains the following.

JoshJava Programming Language: Describes the differences between JoshJava and Java and motivates these differences.

Parsing with Context-Free Look-Ahead-One Grammars: Grammar: Defines what a context-free grammars is give the Expression grammar.

Parsing: Defines what it means to parse a string with respect to a grammar.

Evaluate Expression: Gives and explains the algorithm for parsing and evaluating an expression like (2+3)*v4.

The Grammar for JoshJava: The grammar defining the JoshJava programming language is given.

JoshJava to MARIE Compile: The pre-condition/post-condition contract for each of the procedures in the compiler is given.

The MARIE Assembly Language: The MARIE assembly language is described in detail. In particular what is needed to translate JoshJava code into it. Included are:

- MARIE's architecture
- MARIE assembly language's instruction set
- An Example of Machine Code and Assembly Code
- Data Types
- Physical Memory
- Data Variables
- Instructions for Moving Data
- Instructions for I/O
- Instructions for Actions
- The Program Counter and Jumps
- Conditional Jumps and Ifs
- Constants
- Indirect Addressing
- Marie is Not Easy! It is missing "StoreI X" and bit operations.
- Subroutine Calls

The Compiler Producing MARIE Code: Some of the issues producing MARIE Assembler code are discussed. Memory allocation, boot strapping, subroutines and recursion.

Multiplying and Dividing: We multiply $X \times Y$ and divide X/Y two n bit positive integers X and Y using only $\mathcal{O}(n)$ adding, subtracting, if, and while operations.

Test JoshJava Program: Outlines the JoshJava Programs that we tested our compiler on.

Other Files Produced: This is a list of file that will be handed in.

Token.Java: This is a test program to make sure that the tokenizer is correctly tokenizing our JoshJava code.

Evaluate.Java: Parses and evaluates expressions like (2+3)*v4 producing 20 when v4 is 4. Each line of the input is an expression. The output is the evaluation of each.

evaltest.exp: Is a file containing examples of expressions to be evaluated

evaltest.out: The output from these tests.

Compile.Java: Parses and complies JoshJava code into well documented MARIE assembler code.

Testing JoshJava: The following are examples of JoshJava code to be compiled.

Sorter:

2 JoshJava Programming Language

We started by restricting JoshJava quite a bit so that it could easily parsed to assembler code by a *memoryless one pass parser/compiler*. We were later surprised that very few restrictions were really needed.

A Two-Pass Compiler: Compilers generally need to pass twice through the code. The first time determines the names of all of the variables and the scopes within the program where they are defined. The compiler then decides where in physical memory each variable will be stored and builds a "symbol table" to list this. The second pass of the compiler uses the context free grammar to parse the program into assembler code.

A Memoryless One-Pass Compiler: Both for ease of programming and for aesthetic reasons, we wanted our compiler to be a one pass memoryless parser, meaning that it simply reads through the Java program once, parses it using a grammar, while spewing out the corresponding machine code. In order to avoid the first pass, we first restricted JoshJava so that the resulting symbol table would be fixed.

Initial Simplifications:

- All procedures (other than main) have names $p0, p1, ..., p9$.
- a is always the only global variable. It is an array of a 16 integers.
- $v0, ..., v9$ are always the only local variable for every procedure. They are integers.
- All procedures (other than main) return an integer.

Final Simplifications: In the end, we have very few restrictions.

Integer Variables: To avoid needing to remember the type of a variable, all variables are either of type integer, array of integers, or procedure returning an integer. These are differentiated because arrays references $a[2]$ consist of a string followed by a '[' and procedure calls $p(x, 2)$ consist of a string followed by a '('.

Global vs Local: Global variables need to start with a capital and local variables with a small letter. This allowed us to make the local variables unique by concatenating the variable name with the procedure name.

Block Variable Declarations: All local variables must be declared within a block at the beginning of the procedure declaration. The parser could have handled variable declaration through out the code, but in order for the procedure to recurse, it needs to copy its local variables to the stack. For this reason, we wanted all of its local variable to be in one block.

Lazy: We were too lazy to implement reals, string manipulation, and records.

Commands: We implemented the following commands.

Assignments: v3 = v4+a[7]; a[v2+2*v3] = v9;

If: if(v2 <= v8) .. else ..

While: while(v2 <= v8) ..

Input: in.nextInt()

Output: System.out.println("Hello World" + (v3+v4));

Procedure Calls: p1(v4+a[7], a[v2+2*v3])

Procedure Returns: return v4+a[7];

Data Types:

Integer Variables: All variables are of type integer.

Boolean Expressions: Booleans expressions are implemented to be used within if and while statements, but not to be assigned to a variable.

Constant Strings: Strings are implemented to be use within System.out.println("Hello World") but not to be assigned to a variable or manipulated.

Hex to ASCII: The hard part about implementing strings was not outputting "Hello World", but once MARIE was told to output ASCII instead of Decimals, we had to write a program that would convert our integers to ASCII. When the MARIE code is running this is by far the slowest task. For this reason, the compiler has a second mode, in which it produced MARIE code that skips the "Hello World" strings and outputs the integers as Decimals.

Operations:

Mult,Div,Add,Subt: Complex expressions on integers can be constructed like v3 * (v2+7). We had to implement multiplication and division in MARIE because these were not primitive operations.

Comparisons: Integer comparisons with == ! = < <= > and >= are implemented.

And,Or: Complex expressions on booleans can be constructed like

```
v2 <= v8 && ( v3 = 0 || v4 < v2 ).
```

Scoping of Variables: Variables come in the following forms.

Static Global Variables: The compiler, when seeing anywhere in the Java code the lines

```
static int x = 5;
static int[] a = new int[16];
```

can allocate memory for these variables right there within the MARIE code

```
       Jump L328 / Execution must skip the data
x,     Dec 5     / Memory cell storing the value of x.
a,     Hex ??    / Address of a[0]
       Dec 0     / Memory cell storing the value of a[0].
       Dec 0     / Memory cell storing the value of a[1].
 ...
L328:  Clear
```

These variables are *static* because these variable names can never be used again, otherwise the MARIE assembler code to MARIE machine code assembler will complain that the same label was used more than once.

Assignments: The compiler, when seeing anywhere in the Java code the lines

```
x = y;
```

can implement this with the MARIE code

```
Load y
Store x
```

Because the compiler is memoryless, it does not check that these variables have been allocated.

Assesses like a[i] = b[i]; require indirected addressing. Indirect addressing is conceptually harder and is particularly hard using MARIE's limited command set.

The Scoping of Non-Static Local Variables: If a variable is local to a procedure, then the main difference is that it can only be accessed within this procedure and other procedures can reuse the name. This is implemented by having the memoryless compiler cheat and remember which procedure is currently being compiled and tacking the procedure name p2 after the variable name x producing the unique memory location label x_p2. The challenge is that when the variable is being used like in x=y, the compiler has to also remember which variables are static global and which are non-static local. We solve this by requiring all variable of the first type to start with a capital.

Passing Values into a Procedure: A standard compiler, when compiling the Java code for a procedure declaration

```
public static int p2( int x, int y ) { ...
```

must remember that the first parameter is x and the second is y, so that when it compiles the corresponding calling of the procedure

```
u = p2( 4,5 );
```

it knows that the value 4 needs to be copied into the variable x and the 5 into y. In fact, the compiler must make two passes because the procedure call may occur before before the procedure declaration.

Our compiler, however, is one pass and memoryless. When translating the calling procedure "u = p2(4,5);" into MARIE code, the compiler has access to the values 4 and 5 to be copied but does not know the variable names x and y. On the other hand, when translating the called procedure "p2(int x, int y)" into MARIE code, the compiler has access the variable names x and y, but does not know the values to copy there. Who then does the copying?

At first we thought that this meant that the variable names used for the parameter passing variables would need to be restricted to being v0, v1, ..., v9, vA, .., vE.

However, we then had the insight that this could be done in two steps. The MARIE code will have fixed data cells labeled input0, input1, ..., imputE, and Return. The calling procedure will evaluate the values 4 and 5 and copy them into the fixed locations input0 and input1. The MARIE code then jumps to the called procedure. The call procedure will copy these values from the fixed locations input0 and input1 into its local variables x_p2 and y_p2. When the command "return 7" is found, the 7 is copied to the fixed location Return and the MARIE procedure returns to the calling procedure. The calling procedure then copies the value from Return into its local variable u_p1.

Recursion and the Stack: Implementing Recursion complicates memory allocation and procedure calls even further. For a procedure to be able to call itself, each execution of the routine needs to be able to have its own values in its local variables. Hence, each needs its own *stack frame* within which to allocate is own copy of its local variables. This means that a local variable like x can

no longer be stored in fixed location with the assembler code "x, Dec 4" but needs to addressed with indirect addressing to where this copy is stored within the current stack frame.

We did not want to worry about this when compiling procedures and even procedure calls. Hence, we allocate memory for a procedure's local variables in a fixed block of data cells at the beginning of the code for the procedure. Later when implementing recursion calls, the calling MARIE procedure needs to worry about other executions of the same procedure stepping on its values. Hence, it is designed to copy the values of its local variables in these fixed locations out to a new stackframe. When this called procedure returns, the calling procedure's values are recovered from its stack frame and restored in the fixed memory locations.

To make this coping easier, we insist that all local variables are declared within a block at the beginning of the procedure declaration.

3 Parsing with Context-Free Look-Ahead-One Grammars

An important computer science problem is parsing a string according a given context-free grammar. A *context-free grammar* is a means of describing which strings of characters are contained within a particular language. It consists of a set of rules and a start *non-terminal* symbol. Each rule specifies one way of replacing a non-terminal symbol in the current string with a string of terminal and non-terminal symbols. When the resulting string consists only of terminal symbols, we stop. We say that any such resulting string has been *generated* by the grammar.

Context-free grammars are used to understand both the syntax and the semantics of many very useful languages, such as mathematical expressions, JAVA, and English. The *syntax* of a language indicates which strings of tokens are valid sentences in that language. The *semantics* of a language involves the meaning associated with strings. In order for a compiler or natural language "recognizers" to determine what a string means, it must *parse* the string. This involves deriving the string from the grammar and, in doing so, determining which parts of the string are "noun phrases", "verb phrases", "expressions", and "terms."

Some context-free grammars have a property called *look ahead one*. Strings from such grammars can be parsed in linear time by what I consider to be one of the most amazing and magical recursive algorithms. This algorithm is presented in this chapter. It demonstrates very clearly the importance of working within the friends level of abstraction instead of tracing out the stack frames: Carefully write the specifications for each program, believe by magic that the programs work, write the programs calling themselves as if they already work, and make sure that as you recurse, the instance being input gets "smaller."

The Grammar: We will look at a very simple grammar that considers expressions over $\times$ and $+$. In this grammar, a *factor* is either a simple integer or a more complex expression within brackets; a term is one or more factors multiplied together; and an expression is one or more terms added together. More precisely:

exp ⇒ term
⇒ term + term + ... + term

term ⇒ fact
⇒ fact * fact * ... * fact

fact ⇒ int
⇒ v0 | v1 | v2 | ... | v9
⇒ (exp)

Having more than one ⇒ implies that exp can either be *term* or $term + term$ or $term + term + term$ and so on. Having a | means that fact can either be the variable v0 or v1 or v2 and so on.

Non-Terminals, Terminals, and Rules: More generally, a grammar is defined by a set of *non-terminals*, a set of *terminals*, a *start non-terminal*, and a set of rules. Here the non-terminals are "exp", "term",

and "fact." The terminals are integers, the character "+", and the character "*." The start non-terminal is "exp." Above is the list of rules for this grammar.

A Derivation of a String: A grammar defines the *language* of strings that can be derived in the following way. A derivation of a string starts with the start symbol (a non-terminal). Then each rule, like those above, say that you can replace the non-terminal on the left with the string of terminals and non-terminals on the right.

A Parsing of an Expression: Let s be a string consisting of terminals. A parsing of this string is a tree. Each internal node of the tree is labeled with a non-terminal symbol, the root with the start non-terminal. Each internal node must correspond to a rule of the grammar. For example, for rule $A \Rightarrow BC$, the node is labeled A and its two children are labeled B and C. The leafs of the tree read left to right give the input string s of terminals. The following is an example.

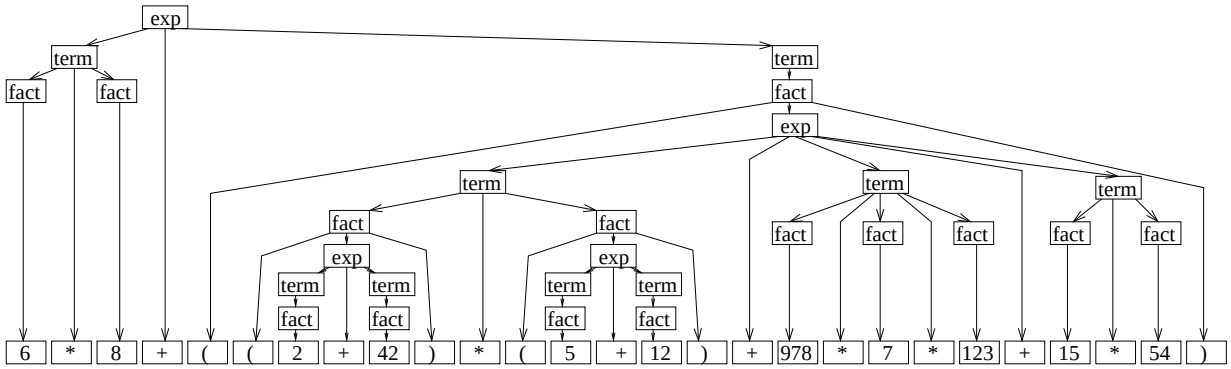

Figure 1: A parse tree for the string $s = 6 * 8 + ((2 + 42) * (5 + 12) + 987 * 7 * 123 + 15 * 54)$.

The Output of the Parser: The implementer might not want the output of the parser to be a tree as shown above. He can make small changes to the parsing code to change the structure of the parsing produced. Here are some of his options.

Parse Tree: For example, if the implementer wants a parsing to be a binary tree representing the expression, then $p_1 + p_2$ would be the operation of constructing a binary tree with the root being a new '+' node, the left subtree being the binary tree p_1, and the right subtree being the binary tree p_2.

The Parsing Abstract Data Type: One option is to write the code description without fulling specifying what a parsing is. This would only say the following: When p is a variable of type parsing, we will use "p=5" to indicate that it is assigned a parsing of the expression "5". We will go on to *overload* the operations $*$ and $+$ as operations that join two parsings into one. For example, if p_1 is a parsing of the expression "2*3" and p_2 of "5*7", then we will use $p = p_1 + p_2$ to denote a parsing of expression "2*3 + 5*7".

Evaluation: On the other hand, if the implementer wants a parsing to be simply an integer evaluation of the expression, then $p_1 + p_2$ would be the integer sum of the integers p_1 and p_2.

Compiler: Finally, if the implementer wants a parsing compiled MARIE assembly code, then p_1 and p_2 would be blocks of code that compute the term and stores the result in a specified memory cell and then $p_1 + p_2$ would be would be blocks of code that adds these two resulting integers and stores the result in a specified memory cell.

Tokenizer: The first step in parsing is to break the input string into tokens. For example the string is

"if(v3 >= 100) then v3 = v3 - 10; /* Comment */",

the the tokens would be

"if" "(" "v3" ">=" "100" ")" "then" "v3" "=" "v3" "-" "10" and ";".

Luckily, Java has a library *java.io.StreamTokenizer* that does just this. The Java object *code* keeps track of where in the file the tokenizer is and the command *code.nextToken*() returns the next token, converting it to an string, character, or real as needed.

The tolenizer even automatically ignores Java comments. The only complications are that we needed to use the command *code.ordinaryChar('-');* so that it parses "- 10" as two tokens "-" and "10". The only thing it does wrong is that it parses ">=" as two tokens. But we can deal with this.

Specifications for the Parsing Algorithm:

Precondition: The input is a sequence of tokens.

Postcondition: If this sequence forms a valid "expression" generated by the grammar, then the output is a parsing of the expression. In this case, the output is an integer evaluation.

The algorithm consists of one routine for each *non-terminal* of the grammar: *GetExp*, *GetTerm*, and *GetFact*.

Specifications for *GetExp*:

Precondition:

***code*:** The global object *code* feeds the parser the sequence of tokens. It keeps track of where in sequence it is and the command *code.nextToken*() returns the next token.

***start-token*$_{exp}$:** The global variable *token* contains the latest token read. We will denote this by *start-token*$_{exp}$.

Postcondition:

***p*$_{exp}$:** The output consists of a parsing p_{exp} of the longest substring starting from the current token and is a valid expression.

***end-token*$_{exp}$:** When it returns the global variable *token* will have been moved to the token that comes immediately after the parsed expression. We will denote this by *end-token*$_{exp}$.

Error: If there is no valid expression starting at $s[i]$, then an error message is output.

The specifications for *GetTerm* and *GetFact* are the same as for *GetExp*, except that they return the parsing of the longest term or factor starting from the current token.

Examples of *GetExp*, *GetTerm*, and *GetFact*: Below are example of what would be returned by each procedure depending on which token is the current token. Here both the current *start-token* and *end-token* are indicated with $\wedge$. The parsing is given with p.

```
GetExp:
   s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
       ^                             ^  p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
         ^                         ^    p =   ( 2 * 8 + 42 * 7 ) * 5 + 8
           ^             ^              p =     2 * 8 + 42 * 7
                 ^       ^              p =             42 * 7
                           ^     ^      p =                       5 + 8

   GetTerm:
   s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
       ^                             ^  p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
         ^                       ^      p =   ( 2 * 8 + 42 * 7 ) * 5
           ^     ^                      p =     2 * 8
                 ^       ^              p =             42 * 7
                           ^ ^          p =                       5
```

```
GetFact:
s = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
    ^                           ^  p = ( ( 2 * 8 + 42 * 7 ) * 5 + 8 )
      ^                 ^           p =   ( 2 * 8 + 42 * 7 )
        ^ ^                         p =     2
                ^   ^               p =             42
                          ^ ^       p =                       5
```

Reasoning for *GetExp*: Consider a sequence of tokens starting at $start\text{-}token_{exp}$. The longest subsequence of tokens that is a valid expression consists of some number of terms added together.

$$\begin{aligned} \text{exp} &\Rightarrow \text{term} \\ &\Rightarrow \text{term} + \text{term} + \ldots + \text{term} \end{aligned}$$

In all of these case, it begins with a term. By magic, assume that the *GetTerm* routine already works. The first token of the term will be the same for that of the expression. Hence, $start\text{-}token_{term_1} = start\text{-}token_{exp}$. Calling *GetTerm*() will return p_{term_1} and $end\text{-}token_{term_1}$, where p_{term_1} is the parsing of this first term and $end\text{-}token_{term_1}$ is the token immediately after this term.

Specifically, if the expression has another term, then $end\text{-}token_{term_1}$ is the '+' that is between these terms. Hence, we can determine whether there is another term by checking $end\text{-}token_{term_1}$. Suppose it is a '+'. The first token of the next term will be the token past the '+'. Hence, *GetExp* will move *token* past the '+' by setting $start\text{-}token_{term_2}$ = the next term after $end\text{-}token_{term_1}$. Then *GetExp* calls *GetTerm* again to get the next term.

Eventually, the returned $end\text{-}token_{term_{last}}$ will not be a '+' but some other character. At this point, *GetExp* has finished reading in all the terms. The first token after the expression is the same as the last after this last term. Therefore, $end\text{-}token_{exp} = end\text{-}token_{term_{last}}$.

As *GetExp* receives these term parsings p_{term_i}, it constructs the expression parsing p_{exp} consisting of all of these term parsings added together, namely $p_{exp} = p_{term_1} + p_{term_2} + \ldots + p_{term_{last}}$.

***GetExp* Code:**

```
public static int GetExp() throws IOException{
    int total;
    total = GetTerm();
    while( true ){
        switch (token) {
          case '+':
              token = code.nextToken(); // move token past the '+'
              total += GetTerm();
              break;
          case '-':
              token = code.nextToken(); // move token past the '-'
              total -= GetTerm();
              break;
          default:
              return total;
        }
    }
}
```

***GetTerm*:** The reasoning and code for *GetTerm* are the same as that for *GetExp* except that *GetTerm* calls *GetFact* and looks for '*' and '/' between these factors.

term ⇒ fact
⇒ fact * fact * ... * fact

Reasoning for *GetFact*: The longest subsequence of tokens that is a valid factor has one of the following two forms:

fact ⇒ int
⇒ v0 | v1 | v2 | ... | v9
⇒ (exp)

Hence, we can determine which form the factor has by testing the first token given by $start\text{-}token_{fact}$.

If $start\text{-}token_{fact}$ is an integer, then we are finished. The parsing p_{fact} is simply this integer and $end\text{-}token_{fact}$ is simply one token past this integer.

If $start\text{-}token_{fact}$ is text, then it must be of the form "v3". We check that the 'v' is there. We set i to be the 3. The parsing p_{fact} is simply the value $v[i]$ of this variable and $end\text{-}token_{fact}$ is simply one token past this "v3".

If $start\text{-}token_{fact}$ = '(', then for this to be a valid factor there must be a valid expression starting at the next token, followed by a closing bracket ')'. We start by moving *token* past the '(' to the beginning of the expression by setting $start\text{-}token_{exp} = start\text{-}token_{fact}$. We can then parse this expression with $GetExp()$, which returns p_{exp} and $end\text{-}token_{exp}$. Because $end\text{-}token_{exp}$ is the first token after the expression, it must be the closing bracket ')'. The first token after the factor is after this ')', giving $end\text{-}token_{fact}$ = the next token after $end\text{-}token_{exp}$. Our parsed factor will be $p_{fact} = (p_{exp})$.

If *token* is neither an integer nor a '(', then it cannot be a valid factor. Give a meaningful error message.

***GetFact* Code:**

```
public static int GetFact() throws IOException{
    int total;
    switch (token) {
      case StreamTokenizer.TT_NUMBER: // A number was found. The value code.nval
          total = (int) code.nval;
          token = code.nextToken();   // Move token past the integer
          return( total );
      case StreamTokenizer.TT_WORD:
             /* code.sval is the token v3.
              * code.sval.charAt(1) is character c=3
              * c-'0' give the integer 3. */
          if(code.sval.charAt(0)!='v') System.out.println("Error: Expected vi");
          int i = code.sval.charAt(1)-'0';
          total = v[i];
          token = code.nextToken();   // Move token past the v3
          return(total);
      case '(':
          token = code.nextToken();   // Move token past the '('.
          total = GetExp();
          if(token!=')') System.out.println("Error: Expected )");
          token = code.nextToken();   // Move token past the ')'.
          return(total);
      default:
          System.out.println("Error: Bad Factor");
          return(0);
    }
}
```

Tree of Stack Frames: $GetExp$ calls $GetTerm$, which calls $GetFact$, which may call $GetExp$, and so on. If one were to draw out the entire tree of stack frames showing who calls who, this would exactly mirror the parse tree that created.

Running Time: The algorithm is very fast flying in linear time though the code spewing out the parsing as it goes.

4 The Grammar for JoshJava:

We now give the formal rules for grammar defining the JoshJava programming language. A program in this language is any string that be generated by the following rules.

Multiple Rules: Having more than one $\Rightarrow$ or having a | imply that nonterminal can be any one of these things.

Look Ahead One: A grammar is said to be *look ahead one* if when there is more then one rule for the same non-terminal, the parser can determine which one to use by testing a single character. This is why we sometimes specify for each rule how the first letter for each rule is distinct.

```
Program          ==>    ProgramBlock
                        ProgramBlock
                          ...
                        ProgramBlock

ProgramBlock     ==>    GListVarDec                                  (begins with "int")
                 ==>    MainProcedure
                 ==>    Procedure

GListVarDec      ==>    GVarDec
                        GVarDec
                          ...
                        GVarDec

GVarDec          ==>    static int GVarName;
                 ==>    static int GVarName = Int;
                 ==>    static int[] GVarName = new int[Int];

Int              ==>    0 | 1 | 2 | 3 | ....

GVarName         ==>    any string starting with a capital letter

GVarName         ==>    any string starting with a capitial letter

MainProcedure    ==>    public static void main (String[] args) {
                          LListVarDec
                          ListCode
                        }

Procedure        ==>    public static int ProcedureName( int LVarName, int LVarName, ...) {
                          LListVarDec
                          ListCode
                        }

ProcedureName    ==>    any string starting with a letter
```

```
LVarName        ==>    any string starting with a small letter

LListVarDec     ==>    LVarDec
                       LVarDec
                         ...
                       LVarDec

LVarDec         ==>    int LVarName;
                ==>    int LVarName = Int;
                ==>    int[] LVarName = new int[Int];

BlockCode       ==>    LineCode
                ==>    { ListCode }

ListCode        ==>    LineCode
                       LineCode
                         ...
                       LineCode                                  (ends with '}')

LineCode        ==>    Assignmentv                            (begins with 'string')
                ==>    Assignmenta                            (begins with 'string[')
                ==>    IfStatement                            (begins with 'if')
                ==>    WhileStatement                         (begins with 'while')
                ==>    ProcedureCall                          (begins with "string(")
                ==>    Return                                 (begins with 'return')
                ==>    OutPut                                 (begins with 'System...')

Assignmentv     ==>    VarName = Exp;

Assignmenta     ==>    VarName[Exp] = Exp;

Exp             ==>    Term
                ==>    Term +|- Term +|- ... +|- Term

Term            ==>    Factor
                ==>    Factor *|/ Factor *|/ ... *|/ Factor

Factor          ==>    Int                                 (begins with '0' .. '9')
                ==>    ( Exp )                             (begins with '(')
                ==>    VarName                             (begins with "string")
                ==>    VarName[Exp]                        (begins with "string[")
                ==>    in.nextInt                          (begins with "in.nextInt")
                ==>    ProcedureCall                       (begins with "string(")

BooleanExp      ==> BooleanTerm
                ==> BooleanTerm || ... || BooleanTerm

 (Note && has higher order of operations just like * does
  true || true && false  = true || (true && false)
                    not   (true || true) && false  )

true            ==>    positive integer
false           ==>    zero
```

```
BooleanTerm     ==> BooleanFactor
                ==> BooleanFactor && ... && BooleanFactor

BooleanFactor   ==> true|false
                ==> BooleanStatement
                ==> (BooleanExp)

BooleanStatement  ==> Exp ==|!=|<|<=|>|>=  Exp

true            ==>    positive integer
false           ==>    zero

IfStatement     ==>    if( BooleanExp )
                         BlockCode
                ==>    if( BooleanExp )
                         BlockCode
                       else
                         BlockCode

WhileStatement  ==>    while( BooleanExp )
                         BlockCode

OutPut          ==>    System.out.println(Factor+"Hello World"+Factor);
                ==>    System.out.print("Hello\nWorld"+Factor);

Procedure       ==>    ProcedureName( Exp, Exp, Exp, ..., Exp )

Return          ==>    return;
                ==>    return Exp;
```

5 The JoshJava Parser

We wrote a complier/parser to translate from JoshJava to well documented MARIE. Just as the Evaluator parser had procedures GetExp, GetTerm, and GetFact, our compiler has a procedure GetT for each (most) non-terminal T. The pre-condition/post-condition contract for these will be as follows.

Parse: For each nonterminal T, the procedure getT parses the longest substring of the code from the indicated starting point that is a valid T.

Output: The output consists of the parsing. For Compile, it is MARIE assembly code that will mirror the actions taken in the block of JoshJava code being parsed.

***token*:** The global variable *token* initially contains the next token to be processed, ie the first token in T. When GetT returns the global variable *token* will be changed to the next token to be processes which is the first token after this parsed T.

Memory Address *m*; GetExp, GetTerm, and GetFact is passed an address m of physical memory. The result of the assembly code produced is to evaluate the Exp, Term, or Factor and store the result in this memory cell.

***nextLabel*:** The global variable *nextLabel* is an integer indicating what the next label in the assembly code should be. For example, if *nextLabel* is 5 then the next label will be "Label5".

***r*:** A parameter integer r is passed to each procedure indicating the depth of the parse tree. For example, when GetProgram will have a value of 0. When it calls GetMainProcedure it will have 1. When it calls List Code it will have 2. And so one. The purpose of this is to indent the MARIE code based on this value.

MARIE comments: This compiler also comments the MARIE instructions to tell which procedure and which rule produced the instruction.

6 The MARIE Assembly Language

MARIE is a very simple assembly language that is used in class rooms. It comes with a graphical simulator/debugger and a graphical Data Path simulator.

MARIE's Architecture: The machine running MARIE has

AC: The accumulator (AC) is special memory cell that numbers are loaded into and out of and adding takes place.

PC: The program counter (PC) is a special memory cell that contains an address/index to the memory cell containing the current instruction being executed. After an instruction is executed, this index is incremented so that it indexes the next instruction. A jump involves changing the PC to the index of a different instruction.

Memory: The rest of memory is an array of integers indexed by 0,1,2,3,...

MARIE Assembly Language's Instruction Set: These commands are all explained in detail below.

Command	Hex	Description
Store X	2	Stores the value from AC into the memory cell addressed by X
Load X	1	Loads the value in the memory cell addressed by X into AC
Clear	A	Put zero in AC
Input	5	Inputs a value from the keyboard into AC
Output	6	Outputs the value in AC to the display
Add X	3	Adds the value in the memory cell addressed by X to AC
Subt X	4	Subtracts the value in the memory cell addressed by X from AC
Jump X	9	Jump to the instruction indexed by the value in memory cell X. Effectively this loads the value at X into PC
Skipcond A	8	if(AC opA 0) then skip the next instruction. 400 is =, 800 is >, 000 is <
AddI X	B	Add Indirect: Uses the value at X as the address of the memory cell whose value is added to AC
JnS X	0	Jump Subroutine: Stores the value in the PC into the memory cell addressed by X and then jumps to cell X+1
JumpI X	C	Jump Indirect: Uses the value at X as the address of the memory cell whose value addresses the instruction to jump to.
Halt	7	Terminates the program
End		End of MARIE code

An Example of Machine Code and Assembly Code: The following chart will be used to explain the relationship between machine code and assembly code and Java

Address	Contents	Label	Assembly Inst	Java
PC	110			program counter
AC	004A			accumulator
			Org 100	
100	5000		Input	AC = in.nextInt();
...				
105	9110		Jump Return	Goto line labeled Return;
...				
110	2112	Return	Store X	Out = AC;
111	7000		Halt	Terminate program
112	001F	X	Dec 31	var X = 31;

Data Types: The computer stores only binary zeros and ones. But these bits can be interpreted in many ways.

Hex: The chart above says that the accumulator currently has the value 004A. This value is written in hexadecimal (Hex), which is an integer base 16. For example, the Hex number 112 is the same as the decimal value $1 * 16^2 + 1 * 16 + 2 * 1 = 274$, A is 10, B is 11, F is 15, and 004A is $0 * 16^3 + 0 * 16^2 + 4 * 16 + 10 * 1 = 79$.

Binary: You may ask why base 16 is used. Well really binary (base 2) is used. Dec 79 in Binary is 01001010. This is hard to read. However, if we break 01001010 into blocks of 4 bits each and translate each into Hex, and put these digits back together, then we get the number in Hex. For example, Binary 0100 is Dec $2^2 = 4$ and Hex 4, Binary 1010 is Dec $2^3 + 2^1 = 10$ and Hex A, hence binary is 01001010 is Hex 4A.

Reals and Ints: We want talk here about how reals and negative integers are encoded in binary.

Char: ASCII code associates every two digit Hex number (8 bits) with an letter of the alphabet. Hex 4A happens to be the letter 'J' for JoshJava. We can do math with these too. For example, 'J'-'A'+1 = Hex 4A - Hex 41 + 1 = 10, because 'J' is the 10th letter in the alphabet.

Memory Addresses: Each memory cell is addressed by an integer 0, 1, 2, and so on. These addresses are also written in hexadecimal. The above chart shows the physical memory cell addressed by Hex 110 contains the value Hex 2112.

Machine Code: Machine code instructions are also just strings of zeros and one. We will later see that the value Hex 2112 is interpreted to be the machine code instruction associated with the assembly instruction "Store X".

Physical Memory: There are no inherent differences between the memory cell addressed by Hex 110 containing the value Hex 2112 and that addressed by Hex 112 containing Hex 001F. It just happens that the value Hex 2112 in the first is interpreted to be the machine code instruction associated with the assembly instruction "Store X" and the value Hex 001F in the second is interpreted to be a data value initiated there by the assembly instruction "X Dec 31".

When assembly code is translated to machine code each line of assembly code is translated either into one machine code instruction or one data value and this is stored in one cell of memory. The command Org says that the first assembly instruction should be stored in the memory cell addressed by Hex 100, the second in 101, and so on. The chart does not list all of the assembly instructions, but suppose that the one "Return Store X" is the 17th. It would then be stored in the 16th memory cell, which is the cell addressed by Hex 100 + Dec 16 = Hex 110. Similarly, the assembly data initiation instruction "X Dec 31" is two lines later and hence is stored in cell Hex 112. To make the code is easier for humans to read, assembly code allows the user to label memory cells. In this case, "Return" is associated with the memory address 110 and "X" with 112.

Data Variables: We have already said that the assembly instruction "X Dec 31" associates the label "X" with memory cell 112. Any value can be stored in this memory cell. Hence, this label "X" effectively

becomes a variable. Dec 31 is not meant to be the last day of the year. It is the decimal integer 31, which is Hex 001F. When the assembly code is translated into machine code, the instruction "X Dec 31" is translated simply into the value Dec 31 = Hex 001F and stored in memory cell 112. This effectively initializes the variable X to have this value. Hence, the assembly instruction "X Dec 31" is equivalent to the Java code "var X = 31".

Instructions for Moving Data: Consider the assembly instruction "Store X". MARIE has fewer than 16 different instructions and they have been assigned a single Hex digit from 0 to F (F=15). The instruction "Store" has been assigned the digit 2. We have already seen that X is associated with 112. Hence, the assembly instruction "Store X" is translated to the machine code instruction 2112 being in memory cell 110. If the program counter (PC) contains this address 110, then the computer will treat this value 2112 as an instruction and execute it. The first digit 2 is interpreted as the instruction "Store the value from AC to a specified memory cell". The remaining three digits 112 are interpreted as the address of the memory cell into which to store the value. Hence, the effect of this instruction is to copy the value 003B from AC to memory cell 112 overwriting the value 001F. In conclusion, the assembly instruction "Store X" is equivalent to the Java code "X = AC;". Similarly, the assembly instruction "Load X" is equivalent to the Java code "AC = X;" and "Clear" is equivalent to "AC = 0;". Values cannot be copied directly from one memory cell to another. They must all go through the AC. Hence, the AC is very busy.

Instructions for I/O: The assembly instruction "Input" reads a value from the keyboard into AC and "Output" outputs the value in AC to the display. In JAVA these might be done with "AC = in.nextInt();" and "System.out.println(AC);". The MARIE Simulator can be set so that its display shows the AC value 004A as Hex 4A, as Dec 74, or Char 'J'.

Instructions for Actions: The assembly instruction "Add X" adds the value in the memory cell addressed by X to AC and "Subt X" subtracts it. In Java, one would write $AC+ = X$ and $AC- = X$. When computers were first built, their instruction set was very sparse because they could not build so many operations into the CPU. As the technology to build CPUs got better, the multiplying instruction was built into the CPU and giving the assembler instruction "Mult X". But later they discovered that simpler CPUs ran faster and hence they put the task of multiplying back into software. When writing our complier from JoshJava we will have to produce this code for multiplying. MARIE's instruction set is EXTREMELY limited. CPUs for example and do bit operation on its values, like shift right, shift left, and test the high and the low bits. Not having this will make fast code for multiplying much harder to write.

The Program Counter and Jumps: The program counter (PC) keeps track of the next machine code instruction to execute. We have already seen that when PC=110, the code "Store X" gets executed. After executing one such instruction, the computer automatically increments the PC and executes the next instruction. In this case, this instruction halts the execution. In order for the program to have "if"s and "loop"s, the execution must be able to jump to another instruction. In our example, if PC=105, then the current machine code is 9110 and the assembly code is "Jump Return". "Return" is a label for the memory address 110. The instruction "Jump" copies this address 110 into PC. Hence, the 110 is the next line to be executed. Structured programmers look down on GOTO commands, but this is what this is.

Conditional Jumps and Ifs: In order for the program to have "if"s and "loop"s, the execution must be able to change its path based on the result of a boolean test. MARIE's only instruction for this is "Skipcond A". As is expected, the only tests allowed will be AC==0, AC>0, and AC<0. Which test done is specified by A.

```
Skipcond 400 is if(AC==0) then skip
Skipcond 800 is if(AC>0)  then skip
Skipcond 000 is if(AC<0)  then skip
```

If the test passes, then the algorithm skips one instruction ahead. The skipped instruction is likely a Jump. For example,

```
The Java code
         if(AC==0)
           AC -= X;
         else
           AC += X;
         Store Y

is implemented as
         Skipcond 400  / If(AC==0) then skip the "Jump Else" instruction
         Jump Else     / If(AC!=0) then Jump to Else
         Subt X        / If(AC==0) then AC -= X  and then Jump to "Store Y"
         Jump EndIf
Else,    Add X         / If(AC!=0) then AC += X and then jump to "Store Y"
Endif,   Store Y
```

Indirect Addressing: We talked about how having the label X associated with the address of a fixed memory cell allows us to think of X as a variable and to think of the assembly instruction "Add X" as being equivalent to the Java code "AC += X;". But what about the JAVA code "AC += A[Y];"? At compile time, we where in memory the array A is stored. In fact, we can let A be a label representing the memory cell that A[0] is stored. However, we cannot know which memory cell A[Y] is stored in we do not unless we know the value of Y, and we wont know that until runtime. During runtime, however, we can compute that A[Y] is stored in the memory cell addressed by the dynamically computed value X=A+Y. But X is itself the address of a memory cell. Hence, the computation must use the value stored in the memory cell addressed by X to the address the memory cell storing A[Y]. Then the value in this memory cell much be added to the AC. The assembly instruction "AddI X" does just this.

```
The Java code
         AC += A[Y];

is implemented as

         Store Temp   / Save the value of AC
         Load A       / Store the address of A[0]
         Add Y        / Increasing this by Y gives you the address of A[Y]
         Store X      / The address of A[Y] is stored in X
         Load Temp    / The original value of AC is returned
         AddI X       / The value in A[Y] is added to AC
         Halt

A,       Hex 300      / Store here the address of where you want A[0]
                      / A block of unused memory cells must follow
                      / address 300
Y,       Hex 5
X,       Hex 0
Temp,    Hex 0
```

Subroutine Calls: Suppose we want to write a subroutine Subr. The label Subr can be the address of the memory cell at which the machine code for the subroutine starts. Then the command "Jump Subr" starts the execution of the subroutine without any problem. But how do you get back to caller of the subroutine when the subroutine returns? We could jump back. The problem the subroutine may be

called from many places, so before we jump to the routine we must store where we jumped from so that we know where to jump back to. The assembly instruction "JnS X" does just this. It Stores the value in PC into memory cell addressed by X and then jumps to the line of code addressed at X+1. This seems strange, but look how elegantly useful it is. Returning from the subroutine requires jumping to an address that is only known dynamically during run time requires indirect addressing again. The assembly instruction "JumpI X" accomplishes this by using the value at X as the address of the memory cell whose value addresses the instruction to jump to. Again think about it yourself before reading on.

```
The Java code
         public static void main (String[] args) {
             int X, Y;
             X = in.nextInt();
             Y = Double( X );
             System.out.println(Y);
         }

         public static int Double( int A ) {
             B = A+A;
             return( B );
         }

is implemented as

main,    Input        / Input a value into X
         Store X
         Store A      / Pass the value of X to the input routine
         JnS Double   / Save the current value of PC and jump to the routine
         Load B       / Get the values returned by the routine
         Store Y
         Output       / Output the value of Y
         Halt
X,       Hex 0
Y,       Hex 0

Double   HEX 0         / Entry Point and Return Address of Subroutine Double
         Load A        / Actual subroutine to double numbers.
         Add A         / AC now holds double the value of A
         Store B       / Store this in B
         JumpI Double  / Return to calling code.
A,       Hex  00
B,       Hex  00
```

7 MARIE is not Easy!

The set of MARIE assembly instructions is very limited. Perhaps too limited. They still have the digits D, E, and F that could have been commands.

Constants: Consider how we could implement the Java code "AC = 20". MARIE does not have an instruction "LoadConstant 20" that puts the value 20 into the accumulator. The instruction "Load X" loads the value of X into AC, but then how does the 20 get into the memory cell for X? The only way is for this 20 to be stored in memory at compile time.

```
The Java code
         X = 20;

Could be implemented as
         Load Const20  / AC = the value 20 in the memory cell Const20
         Store X       / The 20 is put into X
         ...
                       / Bottom of the code.
Const20, Dec 20        / This memory cell will always have the constant 20
                       / in it.
```

The problem with this when compiling JoshJava code into MARIE is that if we put these constants at the bottom of produced the MARIE code, then the compiler would have to remember which constants were used and this would remove some of the joys of pass one parsers. The solution is that data memory cells can be interspersed with code memory cells. Hence, we can put the constant right in place when we need it.

```
The Java code
         X = 20;
         Y = 20;

is implemented as
         Load Label396  / AC = the value 20 in the memory cell Label396
         Jump Label397  / Code jumps over the data cell
Label396 Dec 20         / The data cell containing 20 is in the middle of the code.
Label397 Store X        / Continue the code by storing the 20 in X
         Load Label398  / Repeat for Y.
         Jump Label399
Label398 Dec 20         / The compiler does not remember the last 20,
Label399 Store Y        /  so a different data cell is used.
```

Skip 100 Memory Cells: What if we want two arrays A and B each of size 100. We want A to be the label of the first memory address of the first and B of the second. Do we have to have the command Hex 0 appear 200 times to block off these memory locations? It seems so.

A, Hex Address: The MARIE program that assembles the MARIE assembler instructions into MARIE machine code very kindly counts your instructions for you and allocates each to a memory cell and then allows you to label these memory addresses. This means that you do not worry about memory addresses. Almost. To access a memory cell A[5], you needed to indirectly address into a memory with a command like AddI X. To do this you need the memory cell X to contain an address. One can compute the address of A[5] from A[0] simply by adding 5 to it. But how does one get the address of A[0]? What is needed is the following instruction.

```
A,      Hex B           / Memory cell labeled A contains the address of A[0]
B,      Hex 0           / This cell contains the contents of A[0]
```

MARIE machine code lets you do this without any problem. The problem is that the MARIE assembler instructions do not.

The obvious way of implementing this is to count the number of instructions before this and put the address A+1 in yourself. But if you ever insert an instruction before it, you need to recount.

A kludgey solution uses a MARIE instruction as data. Each MARIE instruction is represented by a single Hex Digit. For example, the instruction "Store" is represented by 2. This instruction takes on argument "Store B", where "B" is the address of the memory cell at which the value in the accumulator

is to be stored. Each memory cell is addressed by three Hex digits. For example, the memory cell labeled B may be addressed by 391. Then the instruction with argument "Store B" combines the one and the three digits into the four digit command 2391. Each memory cell holds four digits that could be interpreted as a command or could be interpreted as data. We want a cell containing the memory address 0392. Working backward, we note that the MARIE instruction represented by 0 happens to be "JnS". Hence, the instruction with argument "JnS B" combines the one and the three digits into the four digit command 0391. We simply interpreted this 0391 as data instead of as a command. In conclusion, the following does what we want.

```
A,      JnS B           / Memory cell labeled A contains the address of A[0]
B,      Hex 0           / This cell contains the contents of A[0]
```

A funny thing that I have used a few times is

```
A,      JnS A           / Memory cell contains its own address
```

"LoadI X" and "StoreI X": Given the importance of this, you would think that MARIE would also have the assembly instructions "LoadI X" and "StoreI X", but it does not!! Note having "LoadI X" is not a problem because it can be implemented with the two instructions "Clear" "AddI X". But not having "StoreI X" is a BIG problem. Without this instruction, how can we implement "A[Y] = AC;"? There must be a way or MARIE would not be very useful. Perhaps one could used the instruction "JumpI X", but this messes with the program counter and we don't want to do this. I came with a way. Try to thing about it yourself and then read the next paragraph.

StoreI X: Store indirect: Uses the value at X as the address of the memory cell into which AC is stored.

Suppose that during run time the memory cell addressed by X currently has the value 112. Then want to store the value of AC into the memory cell addressed by 112. We have a command for this, namely the assembly instruction "Store 112" and the machine code instruction 2112. But how can we use a command that we don't know until run time? ???

The solution is to use the fact that machine code instructions are just numbers and these numbers are stored in memory just like any other values. Hence, we will dynamically rewrite the machine code!

```
The missing assembly instruction
            StoreI X

is implemented as

            Store temp          / Saves AC to be stored later.
            Load X              / X contains the address 112 to get
                                /  the value from
            Add  StoreInst      / Add the command 2 to the front of the
                                /  address producing the machine code 2112
                                /  for "Store 112" which stores the value in
                                /  AC into the cell addressed by 112"
            Store StoreIInst    / Write this machine code where we need to execute it
            Load temp           / Return to AC its original value
StoreIInst, Hex 0               / Execute this machine code to store AC at 112
            Continue the code ...

temp,       Hex    0
X,          Hex  112
StoreInst,  Hex 2000
```

Bit Operations: Another thing MARIE is missing is bit operations, like "What is the first bit of this integer" and "Shift the bits right". These are key for doing things like fast multiplication and division algorithms. Luckily we were able to use Add and Skipcond to write algorithms for these.

More Sparse: If MARIE wanted to have a minimal set of instructions, it could have gotten rid of LOAD, ADDI, JumpI, and JnS using tricks like those above.

8 The Compiler Producing MARIE Code

Some of the issues producing MARIE Assembler code are discussed.

Memory Allocation: We describe here how memory is laid out in the physical machine.

Jump: The first command is to jump past the data to the first command of the code.

Temp: There are temporary variables. For example, if the AC needs to be saved temporarily. (Likely only one will be needed).

Multa,Multb,DecStr: Used by internal subroutines.

StackB,StackP: Address of first cell in the Recursion Stack and next cell in the Recursion stack to push on data.

Input,Return: When a procedure is called, the values being passed in are stored in Input0-InputD by the calling routine and retreaved by the called routine.

***ExpS*: Expression Evaluating Stack:** The routines GetExp, GetTerm, and GetFact are told the address m of a memory cell where it is supposed to write the result of its evaluation. When an expression is to be evaluated, GetExp is first told to use the cell $ExpS$. When GetExp calls GetTerm, GetTerm uses $ExpS+1$. When GetTerm calls GetFact, GetFact uses $ExpS+2$. When GetFact calls GetExp, GetExp uses $ExpS+3$. And so on. We call this the *Expression Evaluating Stack*.

Code: The code goes here.

Recursive Stack: This stores the values of the local variables and the return address for stack of stack frames waiting for a return call.

Boot Strapping: Which came first the chicken or the egg? How did one compile the first compiler? Boot strapping. When the very first computer was built it was programmed by physically attaching wires. When a CPU was built that took machine code as its instructions, the first programs would have to have been written by writing the machine code into memory. This would have gotten tiring really quickly. Hence a program would have been written to translate from assembly code to machine code. When writing in assembly code got tiring, a compiler was written in assembly code to compile a more complex programming language into machine code. Then a compiler would have been written in this more complex programming language to compile an even more complex programming language. And so on.

We did a version of this here. Assembly code for multiplying two values can be found on the web. We used this at first in our compiler to handle multiplications. This code for multiplying, however, is too slow, taking 2^n time for multiplying two n bit. We then wrote a much more complex algorithms in JoshJava that can multiply and divide in n time. We used our compiler to compile this program into assembly code. This assembly code was then used to rewrite our compiler.

Subroutines and Recursion: Generally recursion is implemented by "stacking" a list of stack frames in memory. The stack frame at the top of the stack contains all of the local variables for the procedure call that is currently running. When a procedure is called a new stack frame is added for it on top of the stack. Hence, each of the lower stack frames contains all of the local variables for a procedure call that is poised waiting for a procedure call that it made to return. When the current procedure calls

returns, its stack frame is popped off the stack, allowing the procedure call that called it to return executing.

Because the local variable set of each procedure is fixed, it follows that the amount of physical memory needed for each stack frame is fixed. The advantage of this is that the current stack frame can be stored in a fixed place at the beginning of physical memory, instead of the top of the stack. The advantage of have this location fixed is that all references to variables during the execution of a procedure will be to fixed places in memory. This means that that memory accesses can be done using the assembler code "Store X" which stores the value of the accumulator into the physical memory cell addressed by X. Otherwise, an command indirect addressing command "StoreI X" would be needed that store the value, not in memory cell addressed by X but in the memory cell addressed by the value in the memory cell addressed by X.

This allows us to implement a single procedure call in a way that does not worry about recursive procedure calls. And then recursive procedure calls can be added later if we have time.

Actually, the assembly language MARIE does not even have a "StoreI X" command!

Before a procedure calls another procedure it needs to save recursive stack everything it wants that may get stepped on by the called procedure. This includes:

Procedures Local Variables: The procedure's local variables. These will get stepped on if the procedure calls itself.

ExpS: There is a block of memory called ExpS on which expressions are evaluated. If the command is

```
x = 2+(3*p(1));
```

Then the 2 and the 3 are on the stack when the routine p(1) is called. These need to be saved so that the equation can be evaluated when p(1) returns.

Return Address: When procedure p1 calls p2, the address of the command with p1 to return to when p2 completes is stored at the beginning of the MARIE code for p2. This cell is labeled J_p2. When p2 calls p3, this value within J_p2 needs to be saved on the recursion stack.
*** MORE HERE ***

9 Multiplying and Dividing

The elementary school code for multiplying is too slow, taking 2^n time for multiplying two n bit. The standard high school algorithm only takes n^2 bit operations. However, MARIE is missing is bit operations like "What is the first bit of this integer" and "Shift the bits right". Therefore, we were forced to write our own programs for extracting bits, for multiplication and for division using only Add, Subtract, If, and While.

Our method of getting the bis of Y is as follows. Compute and store 1,2,4,8,16,.. by repeatedly doubling until the value is bigger than Y. Then continually subtract of the biggest of these that is smaller than Y. Keep track of which powers of 2 are subtracted off. This gives the bits of Y. Once we have the bits of Y, we are able to multiply and divide.

Our compiler then compiled programs into MARIE code which was then used by our compiler to compile other JoshJava programs requiring multiplication and division.

Multiply using Adding: Suppose you want to multiply $X \times Y$ two n bit positive integers X and Y. The challenge is that the only operations you have are adding, subtracting, if, and while. The kindergarten algorithm, $4 \times 3 = 3+3+3+3$ takes 2^n time. We want to be able to do it in $\mathcal{O}(n)$ add operations. Note that it takes $\mathcal{O}(n)$ bit operations to add and hence this algorithm will require $\mathcal{O}(n^2)$ bit operations.

The high school algorithm multiplies each bit of X with each bit of Y shifts appropriately and adds. This takes $\mathcal{O}(n^2)$ bit operations as required. The reason we cannot implement this algorithm is that we do not know how to access the bits of X and Y.

We believe that the first step in describing an iterative algorithm is to give the loop invariants, the measure of progress, and the exit condition.

We have values i, x, a, $u[]$, and $v[i]$ such that

LI0': $u[j] = 2^j$, (for all j until $u[j] > X$)

LI0": $v[j] = u[j] \times Y$. Note $v[j] - u[j] \times Y = 0$

LI0: Y does not change.

LI1: $X \times Y = x \times Y + a$

LI2: $x \geq 0$

LI3: $x < 2^i = u[i]$

Measure of progress: i decreases by 1 each iteration.

Exit Condition: $i = 0$

We now need to prove that these loop invariants can be established and maintained and that they together with the exit condition can be used to establish the post condition.

Establishing the Loop Invariant LI0' and LI0":

$u[0] = 1$
$v[0] = Y$
$j = 0$
while($u[j] <= x$)
    $u[j+1] = u[j] + u[j]$
    $v[j+1] = v[j] + v[j]$
    $j = j + 1$
end while

Establishing the Loop Invariant $\langle pre-cond \rangle$ & $code_{pre-loop} \Rightarrow \langle loop-invariant \rangle$:

$code_{pre-loop}$:

$i = j$
$x = X$
$a = 0$

LI1: $x \times Y + a = X \times Y + 0 = X$.

LI2: $x = X \geq 0$ by pre-condition

LI3: $x = x < u[j] = u[i]$ by exit condition of while loop that constructs $u[j]$.

Steps:

loop
    $\langle loop-invariant \rangle$
    exit when $i = 0$
    i = i-1;
    if($u[i] <= x$)
        $x = x - u[i]$
        $a = a + v[i]$
    end if
end loop

Maintaining the Loop Invariant $\langle loop-invariant' \rangle$ & *not* $\langle exit-cond \rangle$ & $code_{loop} \Rightarrow \langle loop-invariant'' \rangle$:

Let x' be the new value and x the old value of x (Same for i and a) There are two cases:

if($u[i] \leq x$): The code in the step gives $i' = i - 1$, $x' = x - u[i']$, and $a' = a + v[i']$.

Maintain LI1: $x' \times Y + a' = (x - u[i']) \times Y + (a + v[i']) = x \times Y - u[i'] \times Y + a + v[i'] = (x \times Y + a) + (v[i'] - u[i'] \times Y) = (X \times Y$ by LI1$) + (0$ by LI0"$) = X \times Y$.

Maintain LI2: By the if statement $u[i'] \leq x$ and hence $x' = x - u[i'] \geq 0$.

Maintain LI3: $x' = x - u[i'] = x - 2^{i-1} < 2^i - 2^{i-1}$ (by LI3) $= 2^{i-1} = 2^{i'} = u[i']$.

else The code in the step gives $i' = i - 1$, $x' = x$, and $a' = a$

Maintain LI1&2: Trivial

Maintain LI3: By the else of the if statement, $u[i'] > x$.

Obtaining the postcondition $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop} \Rightarrow \langle post{-}cond \rangle$:

By exit condition, $i = 0$.

By LI2&3, $0 \leq x < u[i] = 1$, and hence $x = 0$.

By LI1, $X \times Y = x \times Y + a = 0 \times Y + a = a$.

Divide using Adding: Suppose you want to integer divide X/Y two n bit positive integers X and Y, i.e. find values a and r so that $X = a * Y + r$ and $r \in [0, Y)$. The challenge is that the only operations you have are adding, subtracting, if, and while. The kindergarten algorithm, finds 13/3 by continuing to subtract 3 from 13 until it becomes less than 3. This, however, takes 2^n time. We want to be able to do it in $\mathcal{O}(n)$ add operations. Note that it takes $\mathcal{O}(n)$ bit operations to add and hence this algorithm will require $\mathcal{O}(n^2)$ bit operations.

The high school algorithm shifts Y until it is one step from being bigger than X and then subtracts. This is repeated. This takes $\mathcal{O}(n^2)$ bit operations as required. The reason we cannot implement this algorithm is that we do not know how to access the bits of X and Y.

We believe that the first step in describing an iterative algorithm is to give the loop invariants, the measure of progress, and the exit condition.

We have values i, x, a, $u[]$, and $v[i]$ such that

LI0': $u[j] = 2^j$, (for all j until $v[j] > X$)

LI0": $v[j] = u[j] \times Y$. Note $v[j] - u[j] \times Y = 0$

LI0: Y does not change.

LI1: $X = a \times Y + x$

LI2: $x \geq 0$

LI3: $x < (2^i) \times Y = v[i]$

Measure of progress: i decreases by 1 each iteration.

Exit Condition: $i = 0$

We now need to prove that these loop invariants can be established and maintained and that they together with the exit condition can be used to establish the post condition.

Establishing the Loop Invariant LI0' and LI0":

$u[0] = 1$
$v[0] = Y$
$j = 0$
while($v[j] <= x$)
    $u[j+1] = u[j] + u[j]$
    $v[j+1] = v[j] + v[j]$
    $j = j + 1$
end while

Establishing the Loop Invariant $\langle pre{-}cond \rangle$ & $code_{pre\text{–}loop} \Rightarrow \langle loop{-}invariant \rangle$:

$code_{pre\text{–}loop}$:

$i = j$
$x = X$
$a = 0$

LI1: $a \times Y + x = 0 \times Y + X = X$.

LI2: $x = X \geq 0$ by pre-condition

LI3: $x = x < v[j] = v[i]$ by exit condition of while loop that constructs $v[j]$.

Steps:

loop
  $\langle loop{-}invariant \rangle$
  exit when $i = 0$
  i = i-1;
  if($v[i] <= x$)
    $x = x - v[i]$
    $a = a + u[i]$
  end if
end loop

Maintaining the Loop Invariant $\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$:

Let x' be the new value and x the old value of x (Same for i and a)

There are two cases:

if($v[i] \leq x$): The code in the step gives $i' = i - 1$, $x' = x - v[i']$, and $a' = a + u[i']$.

Maintain LI1: $a \times Y + x' = (a + u[i']) \times Y + (x - v[i']) = a \times Y + u[i'] \times Y + x - v[i'] = (a \times Y + x) + (u[i] \times Y - v[i]) = (X$ by LI1$) + (0$ by LI0"$) = X$.

Maintain LI2: By if statement $v[i'] \leq x$ and hence $x' = x - v[i'] \geq 0$.

Maintain LI3: $x' = x - v[i'] = x - 2^{i-1} \times Y < 2^i \times Y - 2^{i-1} \times Y (byLI3) < (2^i - 2^{i-1}) \times Y = 2^{i-1} \times Y = 2^{i'} \times Y = v[i']$.

else The code in the step gives $i' = i - 1$, $x' = x$, and $a' = a$

Maintain LI1&2: Trivial

Maintain LI3: By the else of the if statement, $v[i'] > x$.

Obtaining the postcondition $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop} \Rightarrow \langle post{-}cond \rangle$:

By exit condition, $i = 0$.

By LI2&3, $0 \leq x < v[i] = Y$.

By LI1, $X = a \times Y + x$. Hence $a = X/Y$ with a remainder of x.

```
/*********************************************************************************/
/* Input X,Y >=0
/* Output: X*Y
/*********************************************************************************/

int Mult(int X, int Y) {
    int[] u = new int[16];
    int[] v = new int[16];
    int i,a,b,x;

    u[0]=1;
    v[0]=Y;
    j = 0;
    while( u[j]<=x ) {
        u[j+1] = u[j] + u[j];
```

```
        v[j+1] = v[j] + v[j];
        j = j+1;
    }

    i = j;
    x=X;
    a=0;
    loop
    {
        exit when i=0
        i = i-1;
        if(u[i]<=x) {
            x = x-u[i];
            a = a+v[i];
        }
    }
    return a;
}

/*****************************************************************************************/
/* Input X,Y >=0
/* Output: X/Y
/*   i.e. a and r   so that X=a*Y+r and r in [0,Y).
/*****************************************************************************************/

int Div(int X, int Y) {
    int[] u = new int[16];
    int[] v = new int[16];
    int i,a,b;

    u[0]=1;
    v[0]=Y;
    j = 0;
    while( u[j]<=x ) {
        u[j+1] = u[j] + u[j];
        v[j+1] = v[j] + v[j];
        j = j+1;
    }

    i = j;
    x=X;
    a=0;
    loop
    {
        exit when i=0
        i = i-1;
        if(v[i]<=x) {
            x = x-v[i];
            a = a+u[i];
        }
    }
    return a;
}
```

10 Outline of Steps Taken to Implement the Compiler and tests to make sure it works

```
Commands: We implemented the following commands.
  - Assignments: v3 = v4+a[7]; a[v2+2*v3] = v9;
  - If: if( v2 $<$= v8 ) { .. } else { .. }
  - While: while( v2 $<$= v8 ) { .. }
  - Input: in.nextInt()
  - Output: System.out.println( "Hello World" + (v3+v4) );
  - Procedure Calls: p1( v4+a[7], a[v2+2*v3] )
  - Procedure Returns: return v4+a[7];

Data Types:
  - Integer Variables: All variables are of type integer.
  - Boolean Expressions: Booleans expressions are implemented to be
    used within if and while statements, but not to be assigned to a
    variable.
  - Constant Strings: Strings are implemented to be use within
    System.out.println("Hello World") but not to be assigned to a
    variable or manipulated.
  - Hex to ASCII: The hard part about implementing strings was not
    outputting ``Hello World'', but once MARIE was told to output ASCII
    instead of Decimals, we had to write a program that would convert our
    integers to ASCII. When the MARIE code is running this is by far the
    slowest task. For this reason, the compiler has a second mode, in
    which it produced MARIE code that skips the ``Hello World'' strings
    and outputs the integers as Decimals.

Operations:
  - Mult,Div,Add,Subt: Complex expressions on integers can be
    constructed like v3 * (v2+7). We had to implement multiplication and
    division in MARIE because these were not primitive operations.
  - Comparisons: Integer comparisons with == ! = $<$ $<=$ $>$ and $>=$ are
    implemented.
  - And,Or: Complex expressions on booleans can be constructed like
      v2 <= v8 && ( v3 = 0 || v4 < v2 ).

/*****************************************************************************************

Evaluator
=========

Tasks:
------

- I had to learn how the following grammar can be used to express
expressions of the form (v2+3)*4+7 using the following grammar.

   Exp             ==>    Term
                   ==>    Term +|- Term +|- ... +|- Term

   Term            ==>    Factor
```

```
                  ==>    Factor *|/ Factor *|/ ... *|/ Factor

   Factor         ==>    Int
                  ==>    ( Exp )
                  ==>    VarName

   VarName        ==>    v0 | v1 | ... | v9
```

For example, given the input (v2+3)*4+7
- (v2+3)*4+7 is an expression because is the sum of the terms (v2+3)*4 and 7.
- (v2+3)*4 is a term because is the product of the factors (v2+3) and 3.
- (v2+3) is a factor because is the expression v2+3 in brackets.
- v2+3 is an expression because is the sum of the terms v2 and 3.
- v2 is a term because is a factor.
- v2 is a factor because is a VarName
- v2 is VarName because is one of v0, ... ,v9.

- My father already had code (not in Java) for parsing such expressions. I had rewrite it in Java.

- Then I changed the parser so that the routines GetExp, GetTerm, and GetFact return evaluations of the expression, term, and factor found. For example, given the input v2*3+7

  - How GetFact parses/evaluates v2:
    - It returns the value 2 because the values are set to
      v0 = 0, v1 = 1, v2 = 2, v3 = 3, ..., v9 = 9

  - How GetTerm parses/evaluates v2*3:
    - Asks GetFact to evaluating the first factor v2. It returns 2.
    - Sees the '*' so continues.
    - Asks GetFact to evaluating the second factor 7. It returns 3.
    - Multiplies these and returns 6.

  - How GetExp parses/evaluates v2*3+7:
    - Asks GetTerm to evaluating the first term v2*3. It returns 6.
    - Sees the '+' so continues.
    - Asks GetTerm to evaluating the second term 7. It returns 7.
    - Adds these and returns 13.

- Then I got the program to read in more than one expression separated by ';', printing out the values returned.

- This alone would have been a quite good project!

Test Input:

```
------------
   v3;
   2+v3;
   5*(2+v3)-15;
   (5*(2+v3)-15)*(1+2);
   1+(5*(2+v3)-15)*(1+2)-2;
   10/v5;
```

```
   10/v5*v2;
   5-20;
   -20;
   1+1;

Test Output:
-------------
   Main 3
   Main 5
   Main 10
   Main 30
   Main 29
   Main 2
   Main 4
   Main -15
   Error: Bad Factor 45
   Main 0

Note:
  - The "Main" is there to specify that the main program is printing
    it and not say getTerm.
  - The error occurred because "-20" is a bad expression because it starts with - (ascii 45).
  - Errors: The recovery to the next expression "1+1" did not work, but was ignored.

/*****************************************************************************************

Complier of Assignments of Expression to single variable
========================================================

Tasks:
------

- Hard wired MARIE memory cells for variables v0, v1, ..., v9.
  The code copied from a file called memorycode is:

      v0,  Dec 0000     / 010
      v1,  Dec 0001     / 011
      v2,  Dec 0002     / 012
      v3,  Dec 0003     / 013
      v4,  Dec 0004     / 014
      v5,  Dec 0005     / 015
      v6,  Dec 0006     / 016
      v7,  Dec 0007     / 017
      v8,  Dec 0008     / 018
      v9,  Dec 0009     / 019

  This sets up a memory cell at labeled v0 with initial value 0.
  We commented it by / 010 to remind us that this will be the cell addressed by 0010.

- Then I had to extend the parser/grammar to parse assignments the form v1 = (v2+3)+7;

   Assignmentv     ==>    VarName = Exp;
```

```
- Then I changed the parser so that the routines GetExp, GetTerm,
  and GetFact no longer return evaluations but outputs MARIE code to evaluate them.

  - ExpS0, ExpS1, ..., ExpSF are memory cells that act as a stack for
  evaluating expressions.

- For example, given the input v1 = v2*3+7
  - How GetFact parses/compiles v2 to get MARIE to put the result in ExpS0:
    The code produced is:
            Load v2
            Store ExpS0

  - How GetTerm parses/compiles v2*3 to get MARIE to put the result in ExpS0:
      - Asks GetFact to write code that evaluates the first factor v2 and
        stores it in ExpS0.
      - Sees the '*' so continues.
      - Asks GetFact to write code that evaluates the second factor 3 and
        stores it in ExpS1.
      - Writes code to multiply ExpS0 and ExpS1 and store the result in ExpS0
            ExpS0 = ExpS0 * ExpS1
      - oops MARIE neither has assignments nor multiplication.
        Skip this till later.

  - How GetExp parses/compiles v2*3+7 to get MARIE to put the result in ExpS0:
      - Asks GetTerm to write code that evaluates the first term v2*v3 and
        stores it in ExpS0.
      - Sees the '*' so continues.
      - Asks GetTerm to write code that evaluates the second term 7 and
        stores it in ExpS1.
      - Writes code to add ExpS0 and ExpS1 and store the result in ExpS0
        The code produced is:
            Load  ExpS0
            Add   ExpS1
            Store ExpS0

  - How GetAssignmentv parses/compiles v1 = v2*3+7 to get MARIE to do this assignment.
      - Asks GetExp to write code that evaluates the expression v2*v3+7 and
        stores it in ExpS0.
      - Writes code to move this value to v1.
        The code produced is:
            Load  ExpS0
            Store v1

  - Note the code produced is moves things a little more than it needs to.
            Store  ExpS0
            Load  ExpS0
            Store v1

- This alone would have been a sufficiently hard project!

Test Input:
------------
   v4 = v3;
```

```
   v5 = 2+v3;
   v6 = 5+(2+v3)-7;
   v7 = (5+(2+v3)-7)+(1+2);
   v8 = 1+(5+(2+v3)-7)+(1+2)-2;

Test Output:
-------------
   Surprisingly long MARIE code.

Result of the MARIE code.
-------------------------
In memory:
      v4,  Dec 0003     / 014
      v5,  Dec 0005     / 015
      v6,  Dec 0003     / 016
      v7,  Dec 0006     / 017
      v8,  Dec 0005     / 018


/*********************************************************************************************

Introducing Arrays
==================

Tasks:
------

- Hard wired MARIE memory cells for arrays a[0], a[1], ..., a[9].
  The code copied from a file called memorycode is:

a,      Hex 0030      / 003   Address of a[0]

Dec 0010      / 030   Cell addressed by 030 contains the value 10 of a[0]
Dec 0011      / 031   Cell addressed by 031 contains the value 12 of a[1]
Dec 0012      / 032   ...

  This sets up a memory cell at labeled v0 with initial value 0.
  We commented it by / 010 to remind us that this will be the cell addressed by 0010.

- Then I had to extend the parser/grammar to parse assignments the form a[v2+2*v3] = a[v2+2]+7;

        Assignmenta     ==>    VarName[Exp] = Exp;

- For example, given the input a[v2+2*v3] = a[v2+2]+7;
  - How GetAssignmentv parses/compiles this to get MARIE to do this assignment.
      - Remembers the variable name 'a'.
      - Asks GetExp to write code that evaluates the expression v2+2*v3 and
        stores it in ExpS0.
      - Asks GetExp to write code that evaluates the expression a[v2+2]+7 and
        stores it in ExpS1.
      - Writes code for a[ExpS0] = ExpS1
        The code produced should be:
             Load a          / AC holds address of a[0] (eg 30)
```

```
            Add ExpS0       / Address of a[ExpS0] which is 35
            Store A         / Store the address of a[ExpS0] in A
            Load ExpS1      / Load value to store in a[ExpS0]
            StoreI A        / Store the value in the memory cell addressed by A.

  - But the problem is that MARIE is SO SO simple that is does not an
    the instruction StoreI to indirectly store a value. Hence, my father
    had to figure out a way to get around this. See his document.

Test Input:
------------
   a[1]=a[5];
   a[v9-v7]=7+a[v2];

Test Output:
-------------
   Surprisingly long MARIE code.

Result of the MARIE code.
-------------------------
In memory:
a,      Hex 0030      / 003   Address of a[0]

Dec 0010      / 030   Cell addressed by 030 contains the value 10 of a[0]
Dec 0015      / 031   Cell addressed by 031 contains the value 15 of a[1]
Dec 0019      / 032   Cell addressed by 033 contains the value 19 of a[2]

/*****************************************************************************************

Introducing if and while
========================

Tasks:
------

- Then I implemented Boolean expressions like v2 <= v8 && ( v3 = 0 || v4 < v2 )
  This was very similar to evaluating expressions.

- Then I had to extend the parser/grammar to parse assignments the form if(v1<2) v2=3;

        IfStatement     ==>    if( BooleanExp )
                                 BlockCode
                        ==>    if( BooleanExp )
                                 BlockCode
                               else
                                 BlockCode

        WhileStatement  ==>    while( BooleanExp )
                                 BlockCode

- I found a bug that

    if( v1==1 )
```

```
        v2=2;
    if( v3==1 )
        v2=3;
    else
        v=4;
```

was being compiled as

```
    if( v1==1 )
        v2=2;
        if( v3==1 )
           v2=3;
        else
           v=4;
```

The reason is that we had the faulty grammar

```
    BlockCode     ==>    ListCode
                  ==>    { ListCode }

    ListCode      ==>    LineCode
                         LineCode
                           ...
                         LineCode
```

All was well when this grammar was changed to

```
    BlockCode     ==>    LineCode
                  ==>    { ListCode }
```

Test Input:

```
    v1 = 4;
    if( v1 > 3 && v1 < 5 )
       v2 = 3;
    else {
       v2 = 4;
       v3 = 5;
    }

    if( v1 < 0 || v1 > 5 ) {
       v4 = 3;
     }
    else
       v4 = 4;

    v1 = 1;
    while( v1 < 5 ) {
      a[v1] = 4;
      v1 = v1+1;
    }
```

```
   a[1]=a[5];
   a[v9-v7]=7+a[v2];

Result of the MARIE code.
-------------------------
   As expected.


/*****************************************************************************************************

Introducing Multiplication and Division
=======================================

Tasks:
------

- My father wrote a program in Java to multiply and one to divide.
Normally, these are accomplished by manipulating and shifting the bits
of the numbers. However, MARIE is so simple that it does not have
such instructions. Hence, he had to do it only with addition,
subtraction, if, and while.

- I used the complier that I had written so far to compile his Java
programs for multiplying and dividing into MARIE code.

- Global change and replace changed the labels used in the MARIE code.

- I added the resulting MARIE code to my compiler so that it could
compile JAVA code into MARIE code that multiplies and divides.

Test Input:
------------

    v0 = 16 * 3;
    v1 = 0 * 5;
    v2 = 4 * 5;
    v3 = -8 * 5;
    v4 = 8 * -5;
    v5 = -8 * -5;
    v6 = 34 * 5;
    v7 = 32767 * 5;
    v8 = 32767 * 2;

    v0 = 16 / 3;
    v1 = 0 / 5;
    v2 = 4 / 5;
    v3 = -8 / 5;
    v4 = 8 / -5;
    v5 = -8 / -5;
    v6 = 34 / 5;
    v7 = 32767 / 5;
    v8 = 32767 / 2;
```

```
Result of the MARIE code.
-------------------------
   As expected.


/*****************************************************************************************

Introducing Output
==================

Tasks:
------

- Then I had to extend the parser/grammar to parse output statements

   OutPut    ==>    System.out.println(Factor+"Hello World"+Factor);
             ==>    System.out.print("Hello\nWorld"+Factor);

- The reason the grammar has a Factor and not an Expression in it is because
  5 and (2+5) are factors, while 2+5 is an expression.
    System.out.print("Hello\nWorld"+5); is allowed.
    System.out.print("Hello\nWorld"+(2+5)); is allowed.
    System.out.print("Hello\nWorld"+2+5); is also allowed but means something different.

- Hex to ASCII: The hard part about implementing strings was not
  outputting ‘‘Hello World’’, but once MARIE was told to output ASCII
  instead of Decimals, my father had to write a program that would convert our
  integers to ASCII. When the MARIE code is running this is by far the
  slowest task. For this reason, the compiler has a second mode, in
  which it produced MARIE code that skips the ‘‘Hello World’’ strings
  and outputs the integers as Decimals.

- MARIE did not seem to want to print a blank space, so I replaced it with a ’_’.

Test Input:
-----------

    System.out.print("Hello World\n" + (2+3) + " Really Hi");
    System.out.println(" New line");
    System.out.println("v1 = " + v1);
    System.out.println("a[4] = " + a[4]);
    System.out.println("v3 = " + v3);
    System.out.println("b[2] = " + b[2]);

Result of the MARIE code.
-------------------------
   As expected.


/*****************************************************************************************

Introducing Variable Declarations
=================================
```

```
Tasks:
------

- Then I had to extend the parser/grammar to parse output statements

    LVarName        ==>    any string starting with a small letter

    LListVarDec     ==>    LVarDec
                           LVarDec
                             ...
                           LVarDec

    LVarDec         ==>    int LVarName;
                    ==>    int LVarName = Int;
                    ==>    int[] LVarName = new int[Int];

    LVarName        ==>    any string starting with a small letter
    GVarName        ==>    any string starting with a capital letter

- Different routines can use the same variable name for their local variables.
  Hence, the MARIE memory label used is the local variables names concatenated
  with the procedure name.

Test Input:
------------

    int v1 = 5;
    int[] a = new int[3];

Resulting MARIE code.
----------------------

              Jump S_main  /Jump over data
    v1_main,  Dec 005
    a_main,   JnS L000     /Address of a_main[0]
    L000,     Hex 0        /Memory cell for a_main[0]
              Hex 0
              Hex 0
    S_main,   ...

/*****************************************************************************************

Introducing Procedure Calls
============================

    Program         ==>    ProgramBlock
                           ProgramBlock
                             ...
                           ProgramBlock

    ProgramBlock    ==>    GListVarDec
                    ==>    MainProcedure
```

```
                ==>    Procedure

    MainProcedure   ==>    public static void main (String[] args) {
                             LListVarDec
                             ListCode
                           }

    Procedure       ==>    public static int ProcedureName( int LVarName, int LVarName, ...) {
                             LListVarDec
                             ListCode
                           }

    ProcedureName   ==>    any string starting with a letter
```

- MARIE provides nice instructions for jumping into and out of procedures.

```
    The Java code
         public static void main (String[] args) {
             p1();
         }

         public static int p1() {

             return( B );
         }

    is implemented as

    main,    ...
             JnS p1       / Save the current value of PC in memory cell addressed by p1
                          / and jump to the code cell one after that addressed by p1.
             continue ...

    Double   HEX 0         / Entry Point and Return Address of Subroutine p1
             continue ...

             JumpI p1      / Return to calling code by jumping to the code cell stored in cell p1.
```

- Passing Values into a Procedure:

A standard compiler, when compiling the Java code for a procedure declaration

```
   public static int p2( int x, int y ) { ...
```

must remember that the first parameter is x and the second is y,
so that when it compiles the corresponding calling of the procedure

```
   u = p2( 4,5 );
```

it knows that the value 4 needs to be copied into the variable x and
the 5 into y. In fact, the compiler must make two passes because the
procedure call may occur before before the procedure
declaration.

Our compiler, however, is one pass and memoryless. When translating

the calling procedure ‘‘u = p2(4,5);’’ into MARIE code, the compiler has access to the values 4 and 5 to be copied but does not know the variable names x and y. On the other hand, when translating the called procedure ‘‘p2(int x, int y)’’ into MARIE code, the compiler has access the variable names x and y, but does not know the values to copy there. Who then does the copying?

At first we thought that this meant that the variable names used for the parameter passing variables would need to be restricted to being v0, v1, ..., v9, vA, .., vE.

However, we then had the insight that this could be done in two steps. The MARIE code will have fixed data cells labeled input0, input1, ..., imputE, and Return. The calling procedure will evaluate the values 4 and 5 and copy them into the fixed locations input0 and input1. The MARIE code then jumps to the called procedure. The call procedure will copy these values from the fixed locations input0 and input1 into its local variables x_p2 and y_p2. When the command ‘‘return 7’’ is found, the 7 is copied to the fixed location Return and the MARIE procedure returns to the calling procedure. The calling procedure then copies the value from Return into its local variable u_p1.

Test Goals

- Procedure calls
- Parameter passing
- Returning a value
- Integrating returned value into an expression

Test Input:

```
public static int p1(int x) {
   System.out.println("x = " + x);
   return 5;
}

public static void main (String[] args) {
   int u;
   u = p1(31);
   System.out.println("returned "+ u);
   u = 10+2*p1(32);
   System.out.println("returned "+ u);
}
```

Output of the MARIE code.

```
   x_=_31
   returned_5
   x_=_32
   returned_20
```

```
/*********************************************************************************************
  Larger Test
 =============

Test Goals
------------
  - Larger version of previous
  - Global variables

Test Input:
------------

static int G1 = 8;
static int G2;

public static int p1(int x, int y, int z) {
   int v1 = 9;
   int v2;
   int[] a = new int[5];

   System.out.println("G1 = " + G1 + " G3 = " + G3);
   v1 = 4;
   a[1] = 5;
   a[2] = v1 + a[1];
   System.out.println("x = " + x + " y = " + y + " z = " + z );
   System.out.println("a = " + a[2] + " G2 = " + G2);
   return 5;
}

static int G3 = 2;

public static void main (String[] args) {
   int u;
   G2 = 7;
   u = 10+2*p1(31,32,33);
   G2 = p2(41,42);
   System.out.println("u = " + u);
}

static int G4 = 3;

public static int p2(int u, int y) {
   int v1;

   v1 = 5;
   System.out.println("u = " + u + " y = " + y + " v1 = " + v1 );
   return 45;
}

Output of the MARIE code.
--------------------------
```

```
   G1_=_8_G3_=_2
   x_=_31_y_=_32_z_=_33
   a_=_9_G2_=_7
   u_=_41_y_=_42_v1_=_5
   u_=_20


/*************************************************************************************************
  Implementing Recursion
 ========================

Recursion and the Stack: Implementing Recursion complicates memory
allocation and procedure calls even further. For a procedure to be
able to call itself, each execution of the routine needs to be able to
have its own values in its local variables. Hence, each needs its own
stack frame within which to allocate is own copy of its local
variables. This means that a local variable like x can no longer be
stored in fixed location with the assembler code ``x, Dec 4'' but
needs to addressed with indirect addressing to where this copy is
stored within the current stack frame.

We did not want to worry about this when compiling procedures and even
procedure calls. Hence, we allocate memory for a procedure's local
variables in a fixed block of data cells at the beginning of the code
for the procedure.  Later when implementing recursion calls, the
calling MARIE procedure needs to worry about other executions of the
same procedure stepping on its values.  Hence, it is designed to copy
the values of its local variables in these fixed locations out to a
new stackframe. When this called procedure returns, the calling
procedure's values are recovered from its stack frame and restored in
the fixed memory locations.

To make this coping easier, we insist that all local variables are
declared within a block at the beginning of the procedure declaration.

Test Goals
------------
  - Test Recursion

Test Input:
------------

public static int Fib(int n) {
   int f;
   System.out.println("In:  n = " + n);
   if( n<=1 )
f = n;
   else
f = Fib(n-1) + Fib(n-2);
   System.out.println("Out: n = " + n + " f = " + f);
   return f;
}
```

```
public static void main (String[] args) {
   Fib(5);
}

Expected Answer
---------------

n =  0  1  2  3  4  5
f =  0  1  1  2  3  5

Output of the MARIE code.
-------------------------

   In:__n_=_5
   In:__n_=_4
   In:__n_=_3
   In:__n_=_2
   In:__n_=_1
   Out:_n_=_1_f_=_1
   In:__n_=_0
   Out:_n_=_0_f_=_0
   Out:_n_=_2_f_=_1
   In:__n_=_1
   Out:_n_=_1_f_=_1
   Out:_n_=_3_f_=_2
   In:__n_=_2
   In:__n_=_1
   Out:_n_=_1_f_=_1
   In:__n_=_0
   Out:_n_=_0_f_=_0
   Out:_n_=_2_f_=_1
   Out:_n_=_4_f_=_3
   In:__n_=_3
   In:__n_=_2
   In:__n_=_1
   Out:_n_=_1_f_=_1
   In:__n_=_0
   Out:_n_=_0_f_=_0
   Out:_n_=_2_f_=_1
   In:__n_=_1
   Out:_n_=_1_f_=_1
   Out:_n_=_3_f_=_2
   Out:_n_=_5_f_=_5
```